Amazon Reviews Classification

Angelos Spiliotis
spiliotis@csd.uoc.gr

Eva Perontsi
evaperon@csd.uoc.gr

Antonis Tapanlis
csd3465@csd.uoc.gr

Abstract

The essence of this project, is to try and classify product reviews based on their text body, labeled by ratings, using text pre-processing and supervised learning techniques.

1 Introduction

In the 21st century, the importance of the Internet is beyond doubt. With over 4 billion users worldwide, it affects almost every aspect of everyday life. One of the most successful aspects is commerce. E-commerce is something that even small businesses try to implement by creating e-shops where you can see, order and review products. The idea was to take advantage of the huge amount of data that Amazon has, in form of product reviews and construct a model which can rate reviews automatically. That may be useful to any e-shop that cares about the actual quality that comes from the feedback. So n this project, we tried to predict how good or bad a text review is, derived from any range of products, such as books, apparel, tools, etc. To achieve that, we created a pipeline of models, which do **Natural Language Processing**, exclusively in English, that classifies reviews in five discrete classes.

- Awful ($\star$)
- Bad ($\star\star$)
- Okay ($\star\star\star$)
- Great ($\star\star\star\star$)
- Awesome ($\star\star\star\star\star$)

The data we used to train our model, were text reviews from amazon.com labeled by their star rating. These were separated in categories and sorted by timestamp.

2 The Dataset

Our dataset was the Amazon Customer Reviews [1]which is distributed freely by Amazon and contains all the reviews submitted for amazon products worldwide during the period 1995- 2017. The dataset is available in two formats as parquet and tsv files. We used the tsv files because they were segmented based on product categories and language, while the parquet files were segmented only based on products and that meant that we had to filter out all of those reviews not in english manually. Our data were 30.2GB compressed in the gzip format and after decompression the raw tsv files were approximately 81GB.

As a first step, we only kept 2 out of 15 columns of the dataset, 'amazon_review_body' , 'star_rating' and dropped all the entries that have empty (null) fields.

marketplace	2 letter country code of the marketplace
customer id	Aggregate reviews by a single author
review id	The unique ID of the review
prod. id	The unique Product ID
prod. parent	Aggregate reviews for the same product
prod. title	Title of the product
prod. category	Can be used to group reviews
star rating	**The 1-5 star rating of the review**
helpful votes	Number of helpful votes
total votes	Number of total votes the review received
vine	Written as part of the Vine program
purchase	The review is on a verified purchase
review headline	The title of the review
review body	**The review text**
review date	The date the review was written

Due to some memory issues with AWS we used about half of the dataset. Also our data points where pretty unbalanced where approximately 80% of the reviews where between four and five stars. Our thought was to take random samples from each category and create and other balanced dataset to feed to our pipeline, but we opted out of it because the great disparity in the categories would have as a result a very small dataset.

3 Pipeline

Our first task was to clean the data set and transform it from plain text to meaningful words that will be used as the training set of our classifier. To achieve this, we first used a tokenizer to make our long strings to words, then removed all the unnecessary words and we then fed it to TF-IDF in order to get our numerical features. And finally we used a logistic regression classifier to build a model and make predictions. Every function that we used is implemented by Spark ML library.[2]

3.1 Tokenization

The review body text is a very long string in most cases containing many words, so we needed to split it in order to process it and for this task we used a tokenizer.There are two types of tokenizers implemented by Spark; the simple tokenizer and the regex tokenizer where we can use custom regular expressions as rules to split the text. We opted for the simple tokenizer because we saw that it covered our needs, as we needed a tool, just to split text in words based on space characters and punctuation.

$$[\text{"}This\ is\ a\ plain\ and\ awesome\ string\text{"}] \rightarrow$$

$$[\text{"}This\text{"},\text{"}is\text{"},\text{"}a\text{"},\text{"}plain\text{"},\text{"}and\text{"},\text{"}awesome\text{"},\text{"}string\text{"}]$$

3.2 Stopwords Removal

Now that we have splited successfully our text to tokens, we need to exclude some common words, that appear very frequently and don't give any positive or negative value to a review. So words like "the","or","an","those" are deleted from the token list. In order to do this, we used the StopWordsRemover from Spark ML, which left only the useful and necessary tokens.

$$[\text{"}This\text{"},\text{"}is\text{"},\text{"}a\text{"},\text{"}plain\text{"},\text{"}and\text{"},\text{"}awesome\text{"},\text{"}string\text{"}] \rightarrow$$

$$[\text{"}plain\text{"},\text{"}awesome\text{"},string\text{"}]$$

3.3 TF-IDF

Now that we have successfully filtered out our words we will need to transform them into numbers and for this we used Term Frequency-Inverse Document Frequency(TF-IDF). It's a feature vectorization method used to find the importance of word in a document given a collection of documents. If we use only term frequency it is very easy to get wrong results as we can emphasize in words that very frequent but without any real meaning such as expressions. So with the assumption that if a word appears very frequently in our collection then it shouldn't carry too much meaning we use the inverse document frequency which is a measure of how much meaning a word provides. And we can see it as a mathematic formula below where t denotes each term , d each document which in our case is a review and D the collection of documents which are all of the reviews.

$$TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D)$$

3.3.1 Term Frequency

To find the term frequency $TF(t,d)$ we used the hashingTF method provided by Spark. HashingTF is a transformer which takes sets of terms and converts them into fixed-length feature vectors. A raw feature is mapped to a term using a hash function which in our case is MurmurHash 3 [3] and and term frequencies are calculated using the mapped indices. With this approach we eliminate the need to use global map for our terms which can be computationally expensive, but the drawback is the potential collisions and as a result many raw features may be reduced to the same term. In order to mitigate the collision problem we can increase the target feature dimensions.

3.3.2 Inverse Document Frequency

As we said before if a word appears very often then it means that it doesn't carry too much meaning. So the IDF measures how important a word is.

$$IDF(t,D) = \log \frac{|D|}{DF(t,D)+1}$$

Since a logarithm is used it means that if a word appears to all the reviews then its IDF value becomes zero. Term $|D|$ is the total number of our reviews and also a smoothing term is used to avoid division by zero. IDF is an estimator that means that we fit it with our data and we get an idfModel which then takes feature vectors and scales each feature.

3.4 Logistic Regression

Now that we have our features it is time do a multi-class classification, for this we used the logistic regression which is implemented by Spark ML. Logistic regression supports three optimizers that are implemented by breeze, LBFGS, OWLQN and LBFGSB[4, 5, 6]. All of those are optimization algorithms in the family of quasi-Newton methods used for second order approximations. Given a logistic cost function f with second order approximation we need to minimize this function.

$$f(w+s) \approx f(w) + \nabla f \cdot s^T + \frac{1}{2} \cdot s^T \cdot H(w) \cdot s$$

And then we get our step which is

$$s = -H(w)^{-1} \cdot \nabla f$$

The inverse Hessian matrix H contains all the partial derivatives of f and is defined as $H_{ij} = \frac{\partial^2 f}{\partial w_i \partial w_j}$. This means that it can get very big to fit in memory, that is why we use the above optimization methods in order to get an approximation of the inverse Hessian.

4 Evaluation and Tuning

4.1 Automatic Tuning

To tune our pipeline our pipeline we intended to use ParamGridBuilder and the CrossValidator implemented by Spark ML, but in the end we were forced to do manual tuning due to aws constraints.

4.1.1 ParamGrindBuilder

It is utility provided by Spark that is used to build a parameter grid to search in order to find the optimal parameters for our models. In our case the parameters that we need to search over are the number of features generated by hashingTF and also the parameters of our multi-class logistic regression model the elasticNetParam and regParam. We didn't need to search over the number of iterations that we need for logistic regression due to the fact we are doing second order approximation and the number of iteration that we do are very little 10 for LBFGS optimizer and 100 for OWLQN.

4.1.2 CrossValidator

It is an other utility provided by Spark and is used to find the optimal parameters for our models. CrossValidator splits the input dataset into a set number of folds which serve as separate training and test datasets. Each dataset pair(training,test) uses 2/3 as training and 1/3 as test. Then it evaluates each ParamMap provided by paramGridBuilder by the average evaluation metric for the 3 Models produced by fitting the Estimator on the 3 different (training, test) dataset pairs. The evaluation metric is taken by MulticlassClassificationEvaluator where it evaluates the models based on the prediction accuracy.

4.2 Manual Tuning

In the end we didn't manage to use the above method, so we were forced to do manual tuning of our model. That meant that we had to train separately each model for different parameters and that we couldn't use multiple pairs of training and test sets in order to finish in a reasonable time frame. This is because we would need to supervise our application and take notes over too many hours and we would need to give the data over and over again to IDF and logistic regression manually.

4.3 Evaluation

The metrics that we used to evaluate our pipeline are the following

$$Accuracy = \frac{correct\ predictions}{total\ predictions}$$

$$FPR = \frac{false\ positives}{false\ positives + true\ negatives}$$

$$TPR = \frac{true\ positives}{true\ positives + false\ negatives}$$

$$Fmeasure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

$$Precision = \frac{true\ positives}{true\ positives + false\ positives}$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

Although it's not a very good practice we evaluated our model and did our tuning based mostly on the accuracy. This also was done due to the fact that we trained manually our models.

5 Results

Through the different configurations we run with the logistic regression we show in the graph the most notable results Fig.1. We started with the LBFGS optimizer but due to its poor performace we abandoned it after one run with OWLQN optimizer which as we can see has significantly better results. Although if we had enough time we would have tried some tuning of the LBGS optimizer to see if it would perform any better.

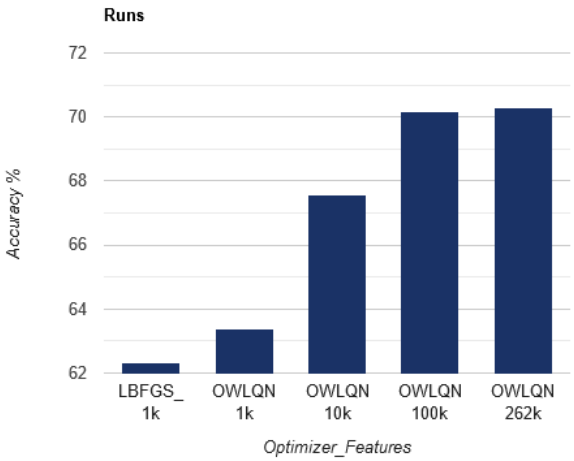

Figure 1: Runs

6 AWS

For our project we used the EC2 and S3 services of AWS provided by Amazon. Amazon provides educate accounts to students for free through a third party company Vocareum which acts as manager of your account and applies some restrictions. We will discuss here the services we used and the problems we had.

6.1 Services

6.1.1 EC2

Elastic Compute Cloud(EC2) is a cluster of computers that are connected together and you can rent them from Amazon. There are many different types computers that you can choose but the main idea is that you tell Amazon what kind of resources you want e.g. number of cores, ram and then Amazon provides that. Although there many different options in terms of VMs we can use, due to the fact that we have an educate account had some limits. The maximum amount of vCPUs we could have was 32 and because there is a correlation between the vCPUs and the memory we could have only 128GB of ram in total. That was a huge problem for us because not only we couldn't load the entire data set but also we couldn't use the CrossValidator to tune our models.

6.1.2 S3

Simple Storage Service(S3) is a distributed storage system made by Amazon where all the data sets are stored and provides an easy way to access them through EC2 cluster and other services.

6.2 Spark Configuration

We ended up using only 4 m5.2xlarge EC2 instances with 8 vCPUs and 32GB of ram each, in contradiction to the 8 we wanted to use due to the constraints of our account. For Spark we didn't use the configuration provided by the Instructor and TA because it was using for the driver an entire node and we deemed it was a waste of resources. So we created executors in all the nodes where each one of them would have 32GB of ram and 8 cores with the exception of the node where the driver resided that we allocated 5GB of ram to the driver and as a result the executor a little less memory than the others.

6.3 AWS Problems

The majority of our problems stemmed from the restrictions applied to our educate account and the fact that we couldn't allocate adequate resources. But another problem arouse from the way that our provided credentials worked. First of all Vocareum provided an **aws_access_key_id**, **aws_secret_access_key** and an **aws_session_token**, in order to access S3 through EC2 you would need all three of those set up in a VM but there was a problem with Spark reading data from S3 through Hadoop. Normaly when you want to read from S3 you give the id and the key to hdfs like this

```
spark.sparkContext.hadoopConfiguration
 .set("fs.s3a.awsAccessKeyId", aws_access_key_id)
spark.sparkContext.hadoopConfiguration
 .set("fs.s3a.awsSecretAccessKey", aws_secret_access_key)
```

The problem here was that id and the key are invlalid without the session token and as far as we saw we couldn't give the token to hdfs in order to access S3 through Spark. The solution to this was to create new VMs and during their configuration we had to set an IAM rule which said that EC2 could access S3 directly without the need of the credentials provided by Vocareum.

6.4 Credits Usage

From the total 90 credits we had as a team we 2/3 of them meaning 60, and although we had around 30 credits left we didn't have any time to run more things. Also as fun fact you can get a negative balance of credits due to the fact that aws doesn't track in real time the usage of credits in educate accounts.

References

[1] Amazon. `https://registry.opendata.aws/amazon-reviews/`.

[2] Apache Spark. `https://spark.apache.org/docs/2.4.0/index.html`.

[3] Søren Dahlgaard, Mathias Knudsen, and Mikkel Thorup. "Practical hash functions for similarity estimation and dimensionality reduction". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6615–6625.

[4] Jorge Nocedal. "Updating quasi-Newton matrices with limited storage". In: *Mathematics of computation* 35.151 (1980), pp. 773–782.

[5] Ciyou Zhu et al. "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization". In: *ACM Transactions on Mathematical Software (TOMS)* 23.4 (1997), pp. 550–560.

[6] Wei XUE and WenSheng ZHANG. "An improved OWL-QN method for solving sparse logistic regression problems". In: *SCIENTIA SINICA Mathematica* 46.1 (2016), pp. 111–121.